CSE 4600 Homework 3 Xv6 Adding Commands and System Calls

Highlights of this assignment:

Xv6 is a teaching operating system developed in 2006 by MIT for teaching operating systems course. For details and installation of it in your own machine, please refer to its web site at:

> https://pdos.csail.mit.edu/6.828/2018/xv6.html

Xv6 is a complete operating system that can boot in Intel i386 machines, which are not widely used nowadays. Often people boot it in the QEMU emulator. The OS is simple with about 7K lines of code and has the same basic internal design as UNIX v6. It does not have many features but has nicely documented source. It is a rewrite of UNIX v6 in ANSI C (standard C) for multicore Intel x86 processors.

1. Compile and Run Xv6

You can copy the source code to your directory by the command,

```
$ git clone https://github.com/mit-pdos/xv6-public.git
```

To compile, execute the command

```
$ make
```

inside *your_directory*. To run it in *qemu* without X-Window,

```
$ make qemu-nox
```

After booting, you can try some commands such as,

```
$ ls
$ echo cse4060
$ cat README
$ grep os README
$ cat README | grep os | wc
$ echo cse 4600 lab report > myFile
$ cat myFile
$ wc README
```

You can exit the OS by typing *ctrl-a c* and quit qemu by typing *quit*. (You can type *ctrl-a c* to get back to the console before quiting qemu.) If you want to debug the kernel, execute

```
$ make qemu-nox-gdb
```

which waits for a gdb (debugger) connection. To connect from debugger, issue the command inside gdb:

```
(gdb) target remote :tcp_port
```

Now you could boot xv6 by the command

```
(gdb) continue
```

and type *crtl-c* to get back to the gdb prompt. You can load the kernel file by

```
(gdb) file kernel
```

You can set the assembly language to i386 by,

```
(gdb) set disassembly-flavor intel
```

and disassemble by

```
(gdb) disass
```

See videos:

- [xv6-1 compile and run OS, and write an application]
- [xv6-2 debugging xv6]

2. Adding Command

Now we implement the file-copy command *cp* for xv6, that can copy a source file to a destination file.
Add the file `cp.c` in xv6 directory.

```
//cp.c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

char buf[512];

int
main(int argc, char *argv[])
{
  int fds, fdd, n;

  if(argc != 3)
  {
    printf(1, "cp SOURCE DEST");
    exit();
  }

  // open source file
  if((fds = open(argv[1], O_RDONLY)) < 0)
  {
      printf(1, "cp: cannot open %s\n", argv[1]);
      exit();
```

```
  }
  // open destination file
  if((fdd = open(argv[2], O_CREATE|O_RDWR)) < 0)
  {
      printf(1, "cp: cannot open %s\n", argv[2]);
      exit();
  }

  while ((n = read(fds, buf, sizeof(buf))) > 0 )
  {
        write(fdd, buf, n);
  }
  close(fds);
  close(fdd);

  exit();
}
```

Modify Makefile

```
$vi Makefile
....
UPROGS=\
        _cat\
        _echo\
        _forktest\
        _grep\
        _init\
        _kill\
        _ln\
        _ls\
        _mkdir\
        _rm\
        _sh\
        _stressfs\
        _usertests\
        _wc\
        _zombie\
        _cp\
....
EXTRA=\
        mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
        ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c  cp.c \
        printf.c umalloc.c\
        README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
        .gdbinit.tmpl gdbutil\

....
....
```

Compile and test it with

```
$make
...
$ cp README myFile
$ ls
$ cat myFile
```

See video [Implementing *cp* in xv6].

3. System Calls

A system call is simply a kernel function that a user application can use to access or utilize system resources. Functions **fork**(), and **exec**() are well-known examples of system calls in UNIX and xv6. We will use a simple example to walk you through the steps of adding a new system call to xv6. We name the system call **cps()**, which prints out the current running and sleeping processes.

An application signals the kernel it needs a service by issuing a software interrupt, a signal generated to notify the processor that it needs to stop its current task, and response to the signal request. Before switching to handling the new task, the processor has to save the current state, so that it can resume the execution in this context after the request has been handled.

The following is a code that calls a system call in xv6 (found in *initcode.S*):

```
.globl start
start:
  pushl $argv
  pushl $init
  pushl $0  // where caller pc would be
  movl $SYS_exec, %eax
  int $T_SYSCALL
```

Basically, it pushes the argument of the call to the stack, and puts the system call number, which is *$SYS_exec* in the example, into *%eax*. All the system call numbers are specified and saved in a table and the system calls of xv6 can be found in the file *syscall.h*.
Next, the code *int $T_SYSCALL* generates a software interrupt, indexing the interrupt descriptor table to obtain the appropriate interrupt handler. The function **trap**() (in *trap.c*) is the specific code that finds the appropriate interrupt handler.

```
// This is the part trap that calls syscall()
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(proc->killed)
      exit();
    proc->tf = tf;
    syscall();
    if(proc->killed)
      exit();
    return;
  }
  .....
}
```

It checks whether the trap number in the generated *trapframe* (a structure representing the processor's state at the time the trap happened) is equal to *T_SYSCALL*. If it is, it calls **syscall**(), the software interrupt handler that's available in *syscall.c*. The function **syscall**() is the final function that checks out *%eax* to obtain the system call's number, which is used to index the table with the system call pointers, and to execute the code corresponding to that system call:

```
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();

  num = proc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    proc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            proc->pid, proc->name, num);
    proc->tf->eax = -1;
  }
}
```

The following are the procedures of adding our exemplary system call **cps()** to xv6.

a. Add name to *syscall.h*:

```
// System call numbers
```

```
#define SYS_fork    1
..........
#define SYS_close  21
#define SYS_cps    22
```

b. Add function prototype to *defs.h*:

```
// proc.c
void            cpuid(void);
......
void            yield(void);
int             cps (void);
```

c. Add function prototype to *user.h*:

```
// system calls
int fork(void);
.....
int uptime(void);
int cps(void);
```

d. Add function call to *sysproc.c*:

```
int
sys_cps(void)
{
  return cps();
}
```

e. Add call to *usys.S*:

```
SYSCALL(cps)
```

f. Add call to *syscall.c*:

```
extern int sys_chdir(void);
.....
extern int sys_cps(void);
.....
static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
.....
[SYS_close]   sys_close,
[SYS_cps]     sys_cps,
};
```

g. Add code to *proc.c*:

```
//current process status
int
cps()
{
  struct proc *p;

  // Enable interrupts on this processor.
  sti();

  // Loop over process table looking for process with pid.
  acquire(&ptable.lock);
  cprintf("name \t pid \t state \n");
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if ( p->state == SLEEPING )
        cprintf("%s \t %d  \t SLEEPING \n ", p->name, p->pid );
```

```
      else if ( p->state == RUNNING )
        cprintf("%s \t %d  \t RUNNING \n ", p->name, p->pid );
  }

  release(&ptable.lock);

  return 22;
}
```

h. Create testing file *ps.c*:

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

int
main(int argc, char *argv[])
{
  cps();

  exit();
}
```

i. Modify *Makefile* to include *ps.c* :

```
$vi Makefile
....
UPROGS=\
        _cat\
        _echo\
        _forktest\
        _grep\
        _init\
        _kill\
        _ln\
        _ls\
        _mkdir\
        _rm\
        _sh\
        _stressfs\
        _usertests\
        _wc\
        _zombie\
        _cp\
        _ps\
        ....
EXTRA=\
        mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
        ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c cp.c ps.c\
        printf.c umalloc.c\
        README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
        .gdbinit.tmpl gdbutil\

....
....
```

After you have compiled and run *"$make qemu-nox"*, you can execute the command "*$ps*" inside xv6. You should see outputs similar to the following:

```
 name    pid     state
init     1       SLEEPING
 sh      2       SLEEPING
 ps      3       RUNNING
```

See video [Adding a system call to xv6 (with caption)].

4. Assignment

1. Do the experiment as described above, that is, add `cp` and `ps` commands and `cps()` system call.
2. Add a new command `touch`, which can create multiple empty files.
3. Add a new system call `date`. This new system call will get the current UTC time and return it to the user program. Here are some hints you may find useful in implementation of the system call `date`
    - Add the new system call and system call number in `syscall.h`
    - `defs.h` is not modified in this task.
    - In `user.h`, add the function prototype `int date(struct rtcdate*);`
    - You may want to use the helper function, `cmostime()` (defined in lapic.c), to read the real time clock. `date.h` contains the definition of the struct `rtcdate` struct, which you will provide as an argument to `cmostime()` as a pointer. The implementation of function `sys_date` in `sysproc.c`

```
int
sys_date (void)
{
    struct rtcdate *d;
    if(argptr(0, (void*)&d, sizeof(struct rtcdate)) < 0)
        return -1;
        cmostime(d);
        return 0;
}
```

    - Update `usys.S` and `syscall.c` accordingly
    - Create a command `date` that calls the new date system call; here's some source you should put in `date.c`:

```
#include "types.h"
#include "user.h"
#include "date.h"

int
main(int argc, char *argv[])
{
    struct rtcdate r;

    if (date(&r)) {
        printf(2, "date failed\n");
        exit();
    }

    // your code to print the date in any format you like...

    exit();

}
```

    - Update `Makefile`

Deliverables:

1. the script log file `hw3log.txt` showing the result of `cp`, `ps`, `touch` and `date`commands:
    - `$ls`
    - `$cp README myfile`
    - `$ls`
    - `$ps`
    - `$touch file1 file2 file3`
    - `$date`
2. `touch.c` and `date.c`

3. `proc.c`
4. `Makefile`

This note is from Dr. Tonglai Yu.
